知识地图 | 一起来复习编译技术核心概念与算法

2020-06-19 朱英达

讲述：朱英达
时长 02:31 大小 2.32M

你好，我是学习委员朱英达。

在“预备知识篇”这个模块，宫老师系统地梳理了编译过程中各个阶段的核心要点，目的就是让我们建立一个编译原理的基础知识体系。那到今天为止，我们就学完了这部分内容，迈出了编译之旅中扎实的第一步。不知道你对这些知识掌握得怎样了？

为了复习，也为了检测我们的学习成果，我根据自己的知识积累和学习情况，整理了一张知识大地图，你可以根据这张地图中标记的七大编译阶段，随时速查常用的编译原理概念和关键算法。

如果你也总结了知识地图，那你可以对照着我这个，给自己一个反馈，看看它们之间有哪些异同点，我们可以在留言区中一起交流和讨论。

不过知识地图的形式，虽然便于你保存、携带、速查，但考虑到图中涉及的概念等内容较多，不方便查看和检索。**所以，我还把地图上的知识点，用文字的形式帮你梳理出来了。**你可以对照着它，来复习和回顾编译技术的核心概念和算法的知识点，构建自己的知识框架。

你在学习这些预备知识的过程中，可能会发现，宫老师并没有非常深入地讲解编译原理的具体概念、理论和算法。所以，如果你想继续深入学习这些基础知识，可以根据宫老师在每讲最后给出的参考资料，去学习龙书、虎书、鲸书等经典编译原理书籍。当然，你也可以去看看宫老师的第一季专栏课《编译原理之美》。

在我看来，相较于编译方面的教科书而言，《编译原理之美》这门课的优势在于，更加通俗易懂、与时俱进，既可以作为新手的起步指导，也能够帮助已经熟悉编译技术的工程师扩展视野，我很推荐你去学习这门课。所以，我邀请编辑添加了相应的知识点到《编译原理之美》的文章链接，如果你有深入学习的需要，你会很方便地找到它。

好了，一起开始复习吧！

一、词法分析：根据词法规则，把字符串转换为 Token

核心概念：正则文法

> 正则文法：词法分析工作的主要文法，它涉及的重要概念是正则表达式。
>
> 正则表达式：正则文法的一种直观描述，它是按照指定模式匹配字符串的一种字符组合。
>
> 正则表达式工具：字符串的模式匹配工具。大多数程序语言都内置了正则表达式的匹配方法，也可以借助一些编译工具，自动化根据正则表达式生成字符串匹配程序，例如 C++ 的 Lex/Yacc 以及 Java 的 ANTLR。

具体实现：手工构造词法分析器、自动生成词法分析器

手工构造词法分析器

> 构造词法分析器使用的计算模型：有限自动机（FSA）。它是用于识别正则文法的一种程序实现方案。
>
> 其组成的词法单元是 Token，也就是指程序中标记出来的单词和标点符号，它可以分成关键字、标识符、字面量、操作符号等多个种类。

在实际的编译器中，词法分析器一般都是手写的。

自动生成词法分析器

具体实现思路：把一个正则表达式翻译成 NFA，然后把 NFA 转换成 DFA。

DFA：确定的有限自动机。它的特点是：该状态机在任何一个状态，基于输入的字符，都能做一个确定的状态转换。

NFA：不确定的有限自动机。它的特点是：该状态机中存在某些状态，针对某些输入，不能做一个确定的转换。这里可以细分成两种情况：一种是对于一个输入，它有两个状态可以转换；另一种是存在ε转换的情况，也就是没有任何字符输入的情况下，NFA 也可以从一个状态迁移到另一个状态。

技术难点

首先，你需要注意，NFA 和 DFA 都有各自的优缺点，以及不同的适用场景。

NFA：优点是在设计上更简单直观，缺点是它无法避免回溯问题，在某些极端的情况下可能会造成编译器运行的性能低下。主要适用于状态较为简单，且不存在回溯的场景。

DFA：优点是它可以避免回溯问题，运行性能较高，缺点是 DFA 通常不容易直接设计出来，需要通过一系列方案，基于 NFA 的转换而得到，并且需要占用额外的空间。主要适用于状态较为复杂，或者对时间复杂度要求较为严苛的工业级词法分析器。

其次，你需要了解基于正则表达式构造 NFA，再去进行模式匹配的算法思路。

从正则表达式到 NFA：这是自动生成词法分析器的一种算法思路。它的翻译方法是，匹配一个字符 i —> 匹配“或”模式 s|t —> 匹配“与”模式 st —> 重复模式，如“?”“*”和“+”等符号，它们的意思是可以重复 0 次、0 到多次、1 到多次，注意在转换时要增加额外的状态和边。

从 NFA 到 DFA：NFA 的运行可能导致大量的回溯，所以我们可以把 NFA 转换成 DFA，让字符串的匹配过程更简单。从 NFA 转换成 DFA 的算法是子集构造法，具体的算法思路你可以参考第 16 讲。

二、语法分析：依据语法规则，编写语法分析程序，把 Token 串转化成 AST

核心概念：上下文无关文法

上下文无关的意思：在任何情况下，文法的推导规则都是一样的。

语法规则由 4 个部分组成：一个有穷的非终结符（或变元）的集合、一个有穷的终结符的集合、一个有穷的产生式集合、一个起始非终结符（变元）。符合这四个特点的文法规则就是上下文无关文法。

两种描述形式：一种是巴科斯范式（BNF），另一种是巴科斯范式的一种扩展形式（EBNF），它更利于自动化生成语法分析器。其中，产生式、终结符、非终结符、开始符号是巴科斯范式的基本组成要素。

上下文无关文法与正则文法的区别：上下文无关文法允许递归调用，而正则文法不允许。上下文无关文法比正则文法的表达能力更强，正则文法是上下文无关文法的一个子集。

具体实现：自顶向下、自底向上

一种是自顶向下的算法思路，它是指从根节点逐层往下分解，形成最后的 AST。

递归下降算法：它的算法思路是按照语法规则去匹配 Token 串。优点：程序结构基本上是跟文法规则同构的。缺点：会造成左递归和回溯问题。**注意**，递归下降是深度优先（DFS）的，只有最左边的子树都生成完了，才会往右生成它的兄弟节点。

LL 算法：对于一些比较复杂的语法规则来说，这个算法可以自动计算出选择不同产生式的依据。方法：从左到右地消化掉 Token。要点：计算 First 和 Follow 集合。

另一种是自底向上的算法思路，它是指从底下先拼凑出 AST 的一些局部拼图，并逐步组装成一棵完整的 AST。

自底向上的语法分析思路：移进，把 token 加入工作区；规约，在工作区内组装 AST 的片段。

LR 算法和 LL 算法一样，也是从左到右地消化掉 Token。

技术难点

首先，你需要掌握 LL 算法的要点，也就是计算 First 和 Follow 集合。

其次，你要了解 LL 算法与 LR 算法的异同点。

> LL 算法：优点是较为直观、容易实现，缺点是在一些情况下不得不处理左递归消除和提取左因子问题。
>
> LR 算法：优点是不怕左递归，缺点是缺少完整的上下文信息，编译错误显示不友好。

三、语义分析：检查程序是否符合语义规则，并为后续的编译工作收集语义信息

核心概念：上下文相关文法

> 属性文法：上下文相关文法对 EBNF 进行了扩充，在上下文无关的推导过程中，辅助性解决一些上下文相关的问题。
>
> 注意：上下文相关文法没有像状态图、BNF 那样直观的分析范式。
>
> 应用场景：控制流检查、闭包分析、引用消解等。

场景案例

1. 控制流检查

像 return、break 和 continue 等语句，都与程序的控制流有关，它们必须符合控制流方面的规则。在 Java 这样的语言中，语义规则会规定：如果返回值不是 void，那么在退出函数体之前，一定要执行一个 return 语句，那么就要检查所有的控制流分支，是否都以 return 语句结尾。

2. 闭包分析

很多语言都支持闭包。而要正确地使用闭包，就必须在编译期知道哪些变量是自由变量。这里的自由变量是指在本函数外面定义的变量，但被这个函数中的代码所使用。这样，在运行期，编译器就会用特殊的内存管理机制来管理这些变量。所以，对闭包的分析，也是上下文敏感的。

具体实现：引用消解、符号表、类型系统、属性计算

引用消解

概念解释：引用消解是一种非常重要的上下文相关的语义规则，它其实就是从符号表里查找被引用的符号的定义。

🔗作用域：指计算机语言中变量、函数、类等起作用的范围。对于变量来说，为了找到正确的引用，就需要用到作用域。一般来说，它有两种使用场景，一种是标识符作用域，一种是词法作用域。

🔗符号表

符号表内包含的信息：名称、分类、类型、作用域等。

存储形式：线性表格、层次化表格。

符号表的作用：维护程序中定义的标识符（ID 类 Token），提供给编译器的各个环节进行操作。

建立符号表的过程：整个编译器前端都会涉及到，词法分析阶段将 ID 类 Token 收集到符号表中，语法分析阶段可进行读取和补充，语义分析阶段做引用消解时符号表的作用至关重要。

注意：符号表跟编译过程的多个阶段都相关。

🔗类型系统

类型：在计算机语言里，类型是数据的一个属性，它的作用是来告诉编译器或解释器，程序可以如何使用这些数据。

类型系统：类型系统是一门语言所有的类型的集合，操作这些类型的规则，以及类型之间怎么相互作用的（比如一个类型能否转换成另一个类型）。

类型检查：这是与类型有关的分析和处理工作之一。主要用于对源程序代码中的一些类型不匹配的情况进行隐式类型转换或直接抛错。

🔗子类型：面向对象编程时，我们可以给某个类创建不同的子类，实现一些个性化的功能；写程序时，我们可以站在抽象度更高的层次上，不去管具体的差异。把这里的结论抽象成一般意义上的类型理论，就是子类型。

类型转换：比如说，表达式“`a=b+10`”，如果 a 的类型是浮点型，而右边传过来的是整型，那么一般就要进行缺省的类型转换。

参数化类型 / 泛型：泛型是程序设计语言的一种风格或范式。泛型允许程序员在强类型程序设计语言中编写代码时使用一些以后才指定的类型，在实例化时作为参数指明这些类型。

属性计算

编译器一般会采用属性计算的方法，来计算出每个 AST 节点的类型属性，然后检查它们是否匹配。

属性文法：属性计算的方法，就是基于语法规则，来定义一些属性计算的规则，在做语法解析或遍历 AST 的时候执行这些规则，我们就可以计算出属性值。这种基于语法规则定义的计算规则，被叫做属性文法（Attribute Grammar）。

属性计算：S 属性（综合属性）、I 属性（继承属性）。

形成的数据结构：Annotated AST（带有标注信息的 AST）。

语法制导的翻译：属性计算的特点是会基于语法规则，增加一些与语义处理有关的规则，我们把这种语义规则的定义叫做语法制导的定义，如果变成计算动作，就叫做语法制导的翻译。

四、运行时机制：程序的两种不同的执行模式

通常情况下，程序有两种执行模式：基于物理机、基于虚拟机。

在物理机上运行

举例：C、C++、Golang。

程序运行的原理：基于指令指针寄存器的控制，顺序从内存读取指令执行。

CPU：运行指令的地方。

多种架构：x86、ARM、MIPS、RISC-V、PowerPC 等。

关键构成：寄存器、高速缓存、功能单元。

汇编代码：操作码（让 CPU 执行的动作）、操作数（指令的操作对象，可以是常数、寄存器和某个内存地址）。

内存：执行指令相关的另一个硬件。

1. 代码区：存放编译完成以后的机器码。
2. 静态数据区：保存程序中全局的变量和常量。
3. 栈：适合保存生存期比较短的数据，比如函数和方法里的本地变量。

   重要知识点：栈帧的构成、活动记录、逐级调用过程。

   栈的特点：申请和释放—修改栈顶指针，生存期与作用域相同。

4. 堆：适合管理生存期较长的一些数据，这些数据在退出作用域以后也不会消失。

   重要知识点：通过操作系统 API 手动申请和释放。

   管理机制：自动管理、手动管理。

操作系统：除了硬件支撑，程序的运行还需要软件，这就是运行时系统。

> 定义：除了硬件支撑，程序的运行还需要软件，这些软件叫做运行时系统。
>
> 操作系统：对于把源代码编译成机器码在操作系统上运行的语言来说（比如 C、C++），操作系统本身就可以看做是它们的运行时系统。
>
> 管理 CPU 资源：分时执行。比如，时间片轮转算法，将 CPU 以时钟周期为单元执行多进程任务，实现并发。
>
> 管理内存资源：逻辑内存（系统内核对内存地址的划定）、物理内存（硬件中具体每一个 bit 实际的硬件存储情况）、虚拟内存（基于操作系统内核对内存管理问题的抽象，通常有一部分虚拟内存实际上是存在磁盘上的）。

在虚拟机上运行

举例：Java、Python、Erlang、Lua。

程序运行的原理：虚拟机是计算机语言的另一种运行时系统。虚拟机上运行的是中间代码，而不是 CPU 可以直接认识的指令。

基于栈的虚拟机：指令在操作数栈的栈顶获取操作数（如 JVM、Python 虚拟机）。

　　优点：易于生成代码。

　　缺点：代码数量较多、不能充分利用寄存器。

基于寄存器的虚拟机：类似于物理机，从寄存器取操作数（如 Erlang、Lua、Dalvik、Ignition）。

　　优点与缺点：与栈机相反。

二者的区别：主要在于如何获取指令的操作数。

五、中间代码：运行各种优化算法、代码生成算法的基础

在这门课程中，宫老师主要从用途和层次、解释执行、呈现格式和数据结构等角度来给你讲解 IR 这一关键概念。如果你想要更深入地了解 IR 的特点，理解如何生成 IR 来实现静态编译的语言，你可以去看《编译原理之美》的第 24、25、26讲。

IR 的用途和层次（从抽象层次的角度来划分）

　　第一类用途：基于源语言做一些分析和变换（HIR）。

　　第二类用途：独立于源语言和 CPU 架构做分析和优化（MIR）。

　　第三类用途：依赖于 CPU 架构做优化和代码生成（LIR）。

IR 的解释执行

　　P-code：直接用于解释执行的 IR。由于它与具体机器无关，因此可以很容易地运行在多种电脑上。这类 IR 对编译器来说，就是做编译的目标代码。

　　注意：P-code 也可能被进一步编译，形成可以直接执行的机器码。如 Java 的字节码。

IR 的呈现格式

大部分 IR 没有像源代码和汇编代码那样的书写格式。

大多数的 IR 跟 AST 一样，只是编译过程中的一个数据结构而已，或者说只有内存格式。比如，LLVM 的 IR 在内存里是一些对象和接口。

为了调试的需要，你可以把 IR 以文本的方式输出，用于显示和分析。

IR 的数据结构

第一种：类似 TAC 的线性结构。

第二种：树结构。

第三种：DAG- 有向无环图。

第四种：PDG- 程序依赖图。

SSA 格式的 IR

概念：SSA，即静态单赋值。这是 IR 的一种设计范式，它要求一个变量只能被赋值一次。

要点：使用 SSA 的形式，体现了精确的“**使用 - 定义（Use-def）**”关系，并且由于变量的值定义出来以后就不再变化，使得基于 SSA 更容易运行一些优化算法。

注意：现代语言用于优化的 IR，很多都是基于 SSA 的，包括 Java 的 JIT 编译器、JavaScript 的 V8 编译器、Go 语言的 gc 编译器、Julia 编译器，以及 LLVM 工具等。

六、代码分析与优化：优化程序对计算机资源的使用，以提高程序的性能

优化分类

是否与机器有关

机器无关：指与硬件特征无关，比如把常数值在编译期计算出来（常数折叠）。

机器有关：需要利用某硬件特有的特征，比如 SIMD 指令可以在一条指令里完成多个数据的计算。

优化范围

本地优化 / 局部优化：基本块内。

全局优化：函数（过程）内。

过程间优化：跨函数（过程）。

优化方法

1. 把常量提前计算出来

常数折叠：程序里的有些表达式，肯定能计算出一个常数值，那就不要等到运行时再去计算，干脆在编译期就计算出来。比如“x=2*3”可以优化成“x=6” 。

常数传播：如果你一旦知道 x 的值其实是一个常量，那你就可以把所有用到 x 的地方，替换成这个常量。

稀疏有条件的常数传播：基于常数传播，还可以进一步导致分支判断条件转化为常量，导致一个分支的代码不会被执行。

2. 用低代价的方法做计算

代数简化：利用代数运算规则所做的简化，比如“x=x*0”可以简化成“x=0”。

3. 消除重复的计算

拷贝传播：遇到相同引用的变量，拷贝替换为同一个，节省内存到寄存器的操作，以此提升运算速度。

值编号（VN 和 GVN）：把相同的值，在系统里给一个相同的编号，并且只计算一次。

公共子表达式消除（CSE）：也会减少程序的计算次数。比如“x:=a+b”和“y:=a+b”，x 和 y 右边的形式是一样的，就可以让 y 等于 x，从而减少了一次对“a+b”的计算。

部分冗余消除（PRE）：是公共子表达式消除的一种特殊情况。

4. 化零为整，向量计算

超字级并行（SLP）：把基本块中的多个变量组成一个向量，用一个指令完成多个变量的计算。

循环向量化：在循环展开的基础上，把多次计算优化成一个向量计算。

5. 化整为零，各个优化

聚合体的标量替换（SROA）：很多语言都有结构和对象这样的复合数据类型，内部包含了多个成员变量，这种数据类型叫做聚合体（aggregates）。

编译器可以把聚合体的成员变量放在寄存器中进行计算，不需要访问内存。

6. 针对循环，重点优化

归纳变量优化：归纳变量是指在循环体内由循环变量派生出来的变量，其变化是很有规律的，因此可以尝试做强度折减优化。

边界检查消除：在循环体内每次循环都会执行的边界检查代码，将其整合抽离出来，避免每次重复判断。

循环展开：通过把循环次数减少，但在每一次循环里，完成原来多次循环的工作量。

循环向量化：在循环展开的基础上，把多次计算优化成一个向量计算。

重组：在循环结构中，使用代数简化和重组，能获得更大的收益。

循环不变代码外提（LICM）：在循环结构中，如果发现有些代码其实跟循环无关，那就应该提到循环外面去，避免一次次重复计算。

代码提升：在条件语句中，如果多个分支条件里都执行了同一句代码，可将其提升至判断条件之前；如果是在循环体内，还可以继续借助循环不变代码外提优化，进一步提升到循环体之外，从而降低计算量。

7. 减少过程调用的开销

尾调用优化和尾递归优化：尾调用就是一个函数的最后一句，是对另一个函数的调用。如果函数最后一句调用的函数是自己，就称为尾递归。尾调用可以将函数调用栈合并，尾递归还可以转换成循环，从而进一步做一系列针对循环语句的优化工作。

内联：内联也叫做过程集成，就是把被调用函数的代码拷贝到调用者中，从而避免函数调用。

内联扩展：内联扩展跟普通内联类似，也是在调用的地方展开代码。不过内联扩展被展开的代码，通常是手写的、高度优化的汇编代码。

叶子程序优化：叶子程序，是指不会再调用其他程序的函数（或过程）。因此，它也可以对栈的使用做一些优化。

8. 对控制流做优化

不可达代码的消除：根据控制流的分析，发现有些代码是不可能到达的，可以直接删掉，比如 return 语句后面的代码。

死代码删除：通过对流程的分析，发现某个变量赋值了以后，后面根本没有再用到这个变量。这样的代码就是死代码，就可以删除。

if 简化：在讲常量传播时我们就见到过，如果有可能 if 条件肯定为真或者假，那么就可以消除掉 if 结构中的 then 块、else 块，甚至整个消除 if 结构。

循环简化：也就是把空循环或者简单的循环，变成直线代码，从而增加了其他优化的机会，比如指令的流水线化。

循环反转：这是对循环语句常做的一种优化，就是把一个 while 循环改成一个 repeat...until 循环（或者 do...while 循环）。这样会使基本块的结构更简化，从而更有利于其他优化。

拉直：如果发现两个基本块是线性连接的，那可以把它们合并，从而增加优化机会。

反分支：也就是减少程序分支，因为分支会导致程序从一个基本块跳到另一个基本块，这样就不容易做优化。比如，把循环内部的 if 分支挪到循环外面去，先做 if 判断，然后再执行循环，这样总的执行 if 判断的次数就会减少，并且循环体里面的基本块不那么零碎，就更加容易优化。

分析方法

1. **控制流分析**（CFA）： 基于程序的控制语句（如条件语句、循环语句、分支语句和基本块语句等）进行分析，建立对程序执行过程的理解，从而进一步做出优化。
2. **数据流分析**（DFA）：基于数据流分析框架（包含“方向 (D)” “值 (V)” “转换函数 (F)” “初始值 (I)”和“交运算 (Λ)”5 个元素）等方式，建立对程序中数据变化情况的理解，从而进一步做出优化。
3. **依赖分析**：分析出程序代码的控制依赖（Control Dependency）和数据依赖（Data Dependency）关系。这对指令排序和缓存优化很重要。

4. **别名分析**：在 C、C++ 等可以使用指针的语言中，同一个内存地址可能会有多个别名，因为不同的指针都可能指向同一个地址。编译器需要知道不同变量是否是别名关系，以便决定能否做某些优化。

优化方法的重要性和顺序

重要性

对所有语言都重要：循环优化等。

面向对象语言：内联、逃逸等。

函数式语言：尾递归优化等。

顺序

要点：机器无关 - 早期，机器相关 - 后期。

注意：一个优化会导致另一个优化，同一个优化会多遍运行。

七、目标代码生成：编译器最后一个阶段的工作，生成针对不同架构的目标代码，也就是生成汇编代码

生成针对不同架构的目标代码

x86：CISC 指令，Register-Memory 架构。在指令里可以混合使用内存地址和寄存器。

ARM：RISC 指令，Load-Store 架构。在 ARM 的汇编中，从寄存器到内存要使用 str（也就是 Store）指令，而从内存到寄存器要使用 ldr（也就是 Load）指令。在这种架构下，指令被分为内存访问（Load 和 Store）和 ALU 操作两大类，且后者只能在寄存器上操作。

策略：编写“代码生成器的生成器”。也就是把 CPU 架构的各种信息描述出来，基于这些信息生成目标代码的生成器，就像根据语法规则，用 ANTLR、bison 这样的工具来生成语法解析器一样。

生成目标代码时的优化工作

1. 指令选择

做指令选择的原因：生成更精简、性能更高的代码；使得同一个功能可以由多种方式实现。

算法：树覆盖算法、自底向上的重写系统（BURS）

2. 寄存器分配

原理：两个变量，不同时活跃，可以共享寄存器。

算法：图染色算法（优点 - 更优化，缺点 - 速度慢）、线性扫描算法（优点 - 不够优化，缺点 - 速度快）

3. 指令排序

原理：CPU 内部的单元可并行。

实现：基于数据依赖图的 List Scheduling 算法。

4. 窥孔优化

思路：提供一个固定大小的窗口，比如能够容纳 10 条指令，并检查窗口内的指令，看看是否可以优化；然后往下滑动窗口，再次检查优化机会。

调用约定的影响

调用约定：你可以发明自己的调用约定，比如哪些寄存器用来放参数、哪些用来放返回值。但是如果要使用别的语言编译好的目标文件，或者想让自己的编译器生成的目标文件被别人使用，那就要遵守某种特定的 ABI 标准。

Unix 和 Linux 系统的调用约定：System V AMD64 ABI。

ABI：即 Application Binary Interface，应用程序的二进制接口。

整体的处理过程

典型过程：指令选择 -> 指令排序 -> 寄存器分配 -> 指令排序 ->Emit 目标代码

要点：基于 LIR，并且不断 lower。

知识地图

- 词法分析
  - 主要文法
    - 正则文法
      - 正则表达式
        - 概念：正则文法的一种直观描述
        - 作用：用于按照指定模式匹配字符串字符组合
        - 正字表达式工具：字符串的模式匹配工具
  - 具体实现
    - 手工构造词法分析器
      - 计算模型：有限自动机 - FSA
        - 用于识别正则文法的一种程序实现方案
      - 词法单元：Token
        - 也就是指程序中标记出来的单词和标点符号，它可以分成关键字、标识符、字面量、操作符号等多个种类
      - 注意：在实际的编译器中，词法分析器一般都是手写的
    - 自动生成词法分析器
      - 不确定的有限自动机 - NFA
        - 针对某输入，多于一条转换路径
        - 存在ε转换
        - 缺点：存在回溯
      - 确定的有限自动机 - DFA
        - 任意输入，只有一条转换路径
        - 优点：没有回溯
        - 缺点：生成DFA的过程需要耗费资源和空间
      - 具体实现思路：把一个正则表达式翻译成NFA，然后把NFA转换成DFA
        - 从正则表达式到NFA
          - 匹配一个字符i
          - 匹配"或"模式：s | t
          - 匹配"与"模式：st
          - 重复模式
        - 从 NFA 到 DFA
          - 子集构造法
            - 计算ε闭包
            - move：基于输入做迁移
  - 技术难点
    - NFA和DFA有各自的优缺点，以及不同的适用场景
      - NFA在设计上更简单直观，但无法避免回溯，极端情况下性能低下
        - 主要适用于状态较为简单，且不存在回溯的场景
      - DFA通常不容易直接设计出来，需要通过一系列方案基于NFA转换而得到，并且需要占用额外的空间，但避免了回溯，运行性能高
        - 主要适用于状态较为复杂，或者对时间复杂度要求较为严苛的工业级词法分析器
- 语法分析
  - 主要文法
    - 上下文无关文法
      - 含义：在任何情况下，文法的推导规则都是一样的
      - 描述形式
        - BNF，巴科斯范式
          - 基本组成要素：产生式、终结符、非终结符、开始符号
        - EBNF，扩展巴科斯范式，它更有利于自动化生成语法
    - 上下文无关文法与正则文法的区别
      - 上下文无关文法允许递归调用，而正则文法不允许。上下文无关文法比正则文法的表达能力更强，正则文法是上下文无关文法的一个子集
  - 具体实现
    - 自顶向下
      - 递归下降算法：按照语法规则去匹配Token串
        - 优点：程序结构基本上是跟文法规则同构的
        - 缺点：会造成左递归和回溯问题
          - 左递归问题
            - 文法改写方法
            - 解决结合性问题
              - 以循环代替递归
          - 回溯问题
            - 当某个非终结符存在多个产生式，可以先尝试其中一个；不成功的话再回退回来，继续尝试下一个产生式
        - 注意：递归下降是深度优先遍历（DFS）的，只有最左边的子树都生成完了，才会往右生成它的兄弟节点
      - LL系算法
        - 要点：计算First集合和Follow集合
        - 算法思路：提取左公因子，预读更多Token
    - 自底向上
      - 移进和规约
        - 移进：把Token加入工作区
        - 规约：在工作区内组装AST的片段
      - LR系算法
  - 技术难点
    - LL算法的要点：计算First和Follow集合
    - LL与LR的对比
      - 优点：不怕左递归
      - 缺点：缺少完整的上下文信息，编译错误显示不友好
- 语义分析
  - 主要文法
    - 上下文相关文法
      - 属性文法
        - 对EBNF进行了扩充，在上下文无关的推导过程中，辅助性解决一些上下文相关的问题
      - 应用场景
        - 控制流检查
        - 闭包分析
        - 引用消解
      - 注意
        - 上下文相关文法没有像状态图、BNF那样直观的分析范式
  - 场景案例
    - 控制流检查
      - 像return、break 和continue等语句，都与程序的控制流有关，它们必须符合控制流方面的规则
      - 在 Java 这样的语言中，语义规则会规定：如果返回值不是 void，那么在退出函数体之前，一定要执行一个 return 语句，那么就要检查所有的控制流分支，是否都以 return 语句结尾
    - 闭包分析
      - 很多语言都支持闭包，而要正确地使用闭包，就必须在编译期知道哪些变量是自由变量，这里的自由变量是指在本函数外面定义的变量，但被这个函数中的代码所使用
      - 这样，在运行期，编译器就会用特殊的内存管理机制来管理这些变量。所以，对闭包的分析，也是上下文敏感的
  - 具体实现
    - 引用消解
      - 引用消解是一种非常重要的上下文相关的语义规则，它其实就是从符号表里查找被引用的符号的定义
      - 作用域：指计算机语言中变量、函数、类等起作用的范围。对于变量来说，为了找到正确的引用，就需要用到作用域
        - 注意：不同语言，作用域的规则不同
    - 符号表
      - 信息：名称、分类、类型、作用域等
      - 存储形式
        - 线性表格
        - 层次化表格
      - 作用：维护程序中定义的标识符（ID类Token），提供给编译器的各个环节进行操作
      - 建立符号表的过程
        - 整个编译器前端都会涉及到，词法分析阶段将ID类Token收集到符号表中，语法分析阶段可进行读取和补充
      - 注意：符号表跟编译过程的多个阶段有关
    - 类型系统
      - 类型：一组数据及其操作
      - 类型检查
        - 这是与类型有关的分析和处理工作之一，主要用于对源程序代码中的一些类型不匹配的情况进行隐式类型转换或直接抛错
      - 类型系统
        - 是一门语言所有的类型的集合，操作这些类型的规则，以及类型之间怎么相互作用的（比如一个类型能否转换成另一个类型）
      - 子类型
        - 面向对象编程时，我们可以给某个类创建不同的子类，实现一些个性化的功能；写程序时，我们可以站在抽象度更高的层次上，不去管具体的差异。把这里的结论抽象成一般意义上的类型理论，就是子类型

极客时间

- 类型转换
  - 比如说，表达式"a=b+10"，如果 a 的类型是浮点型，而右边传过来的是整型，那么一般就要进行缺省的类型转换
- 参数化类型/泛型：泛型是程序设计语言的一种风格或范式
- 属性计算
  - 编译器一般会采用属性计算的方法，来计算出每个 AST 节点的类型属性，然后检查它们是否匹配
  - 属性文法
    - 属性计算的方法，就是基于语法规则，来定义一些属性计算的规则，在做语法解析或遍历 AST 的时候执行这些规则，我们就可以计算出属性值
      - 这种基于语法规则定义的计算规则，被叫做属性文法
  - S属性（综合属性）、I属性（继承属性）
  - 形成的数据结构
    - Annotated AST（带标注信息的AST）
  - 语法制导的翻译
    - 属性计算的特点是会基于语法规则，增加一些与语义处理有关的规则，我们把这种语义规则的定义叫做语法制导的定义，如果变成计算动作，就叫做语法制导的翻译

编译原理

运行时机制

基于物理机

- 举例：C、C++、Golang
- 程序运行原理
  - 基于指令指针，顺序从内存读取指令执行
- CPU
  - 多种架构：x86、ARM、MIPS、RISC-V、PowerPC...
  - 关键构成：寄存器、高速缓存、功能单元
  - 汇编代码
    - 操作码
      - 让 CPU 执行的动作
    - 操作数：指令的操作对象
      - 立即数
      - 寄存器
      - 内存地址
- 内存
  - 分区：代码区、静态数据区、栈、堆
  - 栈
    - 栈帧的构成、活动记录
    - 逐级调用过程
    - 特点
      - 申请和释放：修改栈顶指针
      - 生存期与作用域相同
  - 堆
    - 通过操作系统API手动申请和释放
    - 被分配的内存的生存期可以超出申请它的代码的作用域
    - 管理机制
      - 手动管理
      - 自动管理（GC）
- 操作系统
  - 管理CPU资源
    - 分时执行
      - 如时间片轮转算法，将CPU以时钟周期为单元执行多进程任务，实现并发
  - 管理内存资源
    - 逻辑内存 vs 物理内存
      - 逻辑内存：系统内核对内存地址的划定
      - 物理内存：硬件中具体每一个bit实际的硬件存储情况
    - 虚拟内存
      - 基于操作系统内核对内存管理问题的抽象，通常有一部分虚拟内存实际上是存在磁盘上的
  - 加载程序并执行

基于虚拟机

- 举例：Java、Python、Erlang、Lua
- 程序运行的原理
  - 虚拟机是计算机语言的另一种运行时系统。虚拟机上运行的是中间代码，而不是 CPU 可以直接认识的指令
- 基于栈的架构
  - 指令在操作数栈的栈顶获取操作数
  - 例子
    - JVM
    - Python虚拟机
  - 优点：易于生成代码
  - 缺点
    - 代码数量较多
    - 不能充分利用寄存器
- 基于寄存器的架构
  - 类似于物理机，从寄存器取操作数
  - 例子
    - Erlang
    - Lua
    - Dalvik
    - Ignition（V8的字节码执行模式）
  - 优点与缺点：与栈机相反

中间代码

IR的用途和层次

- 第一类用途：基于源语言做一些分析和变换
- 第二类用途：独立于源语言和CPU架构做分析和优化
- 第三类用途：依赖于CPU架构做优化和代码生成

IR的解释执行

- P-code
  - 直接用于解释执行的IR
    - 由于它与具体机器无关，因此可以很容易地运行在多种电脑上
    - 这类IR对编译器来说，就是做编译的目标代码
  - 注意：P-code也可能被进一步编译，形成可以直接执行的机器码，如Java的字节码

IR的呈现格式

- 大部分IR没有像源代码和汇编代码那样的书写格式
  - 大多数的IR跟AST一样，只是编译过程中的一个数据结构而已，或者说有内存格式。比如，LLVM的IR在内存里是一些对象和接口
  - 为了调试的需要，你可以把IR以文本的方式输出，用于显示和分析

IR的数据结构

- 第一种：类似TAC的线性结构
- 第二种：树结构
- 第三种：DAG-有向无环图
- 第四种：PDG-程序依赖图

SSA格式的IR

- 概念
  - SSA，即静态单赋值。这是IR的一种设计范式，它要求一个变量只能被赋值一次
- 要点
  - 使用SSA的形式，体现了精确的"使用-定义（Use-def）"关系，并且由于变量的值定义出来以后就不再变化，使得基于SSA更容易运行一些优化算法
- 注意
  - 现代语言用于优化的IR，很多都是基于SSA的，包括Java的JIT编译器、JavaScript的V8编译器、Go语言的gc编译器、Julia编译器，以及LLVM工具等

优化分类

- 是否与机器有关
  - 机器无关
    - 指与硬件特征无关，比如把常数值在编译期计算出来（常数折叠）
  - 机器有关
    - 需要利用某硬件特有的特征，比如 SIMD 指令可以在一条指令里完成多个数据的计算
- 优化范围
  - 本地优化/局部优化：基本块内
  - 全局优化：函数（过程）内

极客时间

- 过程间优化：跨函数（过程）

代码分析与优化

优化方法

- 1.把常量提前计算出来
  - 常数折叠 — 程序里的有些表达式，肯定能计算出一个常数值，那就不要等到运行时再去计算，干脆在编译期就计算出来 — 比如"x=2*3"可以优化成"x=6"
  - 常数传播 — 如果你一旦知道 x 的值其实是一个常量，那你就可以把所有用到 x 的地方，替换成这个常量
  - 稀疏有条件的常数传播 — 基于常数传播，还可以进一步导致分支判断条件转化为常量，导致一个分支的代码不会被执行
- 2.用低代价的方法做计算 — 代数简化：利用代数运算规矩做的简化 — 比如"x=x*0"可以简化成"x=0"
- 3.消除重复的计算
  - 拷贝传播 — 遇到相同引用的变量，拷贝替换为同一个，节省内存到寄存器的操作，以此提升运算速度
  - 值编号（VN和GVN） — 把相同的值，在系统里给一个相同的编号，并且只计算一次
  - 公共子表达式消除（CSE）：也会减少程序的计算次数 — 比如"x := a + b"和"y := a + b"，x和y右边的形式是一样的，就可以让y等于x，从而减少了一次对"a+b"的计算
  - 部分冗余消除（PRE） — 是公共子表达式消除的一种特殊情况
- 4.化零为整，向量计算
  - 超字级并行（SLP） — 把基本块中的多个变量组成一个向量，用一个指令完成多个变量的计算
  - 循环向量化 — 在循环展开的基础上，把多次计算优化成一个向量计算
- 5.化整为零，各个优化 — 聚合体的标量替换（SROA）
  - 很多语言都有结构和对象这样的复合数据类型，内部包含了多个成员变量，这种数据类型叫做聚合体
  - 编译器可以把聚合体的成员变量放在寄存器中进行计算，不需要访问内存
- 6.针对循环，重点优化
  - 归纳变量优化 — 归纳变量是指在循环体内由循环变量派生出来的变量
  - 边界检查消除 — 在循环体内每次循环都会执行的边界检查代码，将其整合抽离出来，避免每次重复判断
  - 循环展开 — 通过把循环次数减少，但在每一次循环里，完成原来多次循环的工作量
  - 循环向量化 — 在循环展开的基础上，把多次计算优化成一个向量计算
  - 重组 — 在循环结构中，使用代数简化和重组，能获得更大的收益
  - 循环不变代码外提（LICM） — 在循环结构中，如果发现有些代码其实跟循环无关，就应该提到循环外面去，避免一次次重复计算
  - 代码提升
    - 在条件语句中，如果多个分支条件里都执行了同一句代码，可将其提升至判断条件之前
    - 如果是在循环体内，还可以可以继续借助循环不变代码外提优化，进一步提升到循环体之外，从而降低计算量
- 7.减少过程调用的开销
  - 尾调用优化和尾递归优化
    - 尾调用：就是一个函数的最后一句，是对另一个函数的调用，尾调用可以将函数调用栈合并
    - 尾递归：如果函数最后一句调用的函数是自己，就称为尾递归，它除了可以将函数调用栈合并以外，还可以转换成循环，从而进一步做一系列针对循环语句的优化工作
  - 内联 — 内联也叫做过程集成，就是把被调用函数的代码拷贝到调用者中，从而避免函数调用
  - 内联扩展
    - 内联扩展跟普通内联类似，也是在调用的地方展开代码
    - 不过内联扩展被展开的代码，通常是手写的、高度优化的汇编代码
  - 叶子程序优化 — 叶子程序，是指不会再调用其他程序的函数（或过程），因此它也可以对栈的使用做一些优化
- 8.对控制流做优化
  - 不可达代码的消除 — 根据控制流的分析，发现有些代码是不可能到达的，可以直接删掉，比如 return 语句后面的代码
  - 死代码删除 — 通过对流程的分析，发现某个变量赋值了以后，后面根本没有再用到这个变量。这样的代码就是死代码，就可以删除
  - if简化 — 如果有可能if条件肯定为真或者假，那么就可以消除掉 if 结构中的then块、else块，甚至整个消除if结构
  - 循环简化 — 也就是把空循环或者简单的循环，变成直线代码，从而增加了其他优化的机会，比如指令的流水线化
  - 循环反转 — 这是对循环语句常做的一种优化，就是把一个 while 循环改成一个 repeat...until 循环（或者 do...while 循环）
  - 拉直 — 如果发现两个基本块是线性连接的，那可以把它们合并，从而增加优化机会
  - 反分支：也就是减少程序分支 — 比如，把循环内部的 if 分支挪到循环外面去，先做 if 判断，然后再执行循环，这样总的执行 if 判断的次数就会减少，并且循环体里面的基本块不那么零碎，更加容易优化

分析方法

- 控制流分析（CFA） — 基于程序的控制语句（如条件语句、循环语句、分支语句和基本块语句等）进行分析，建立对程序执行过程的理解，从而进一步做出优化
- 数据流分析（DFA） — 基于数据流分析框架（包含"方向(D)""值(V)""转换函数(F)""初始值(I)"和"交运算(Λ)"5 个元素）等方式，建立对程序中数据变化情况的理解，从而进一步做出优化
- 依赖分析
  - 分析出程序代码的控制依赖（Control Dependency）和数据依赖（Data Dependency）关系
  - 注意：这对指令排序和缓存优化很重要
- 别名分析
  - 在 C、C++ 等可以使用指针的语言中，同一个内存地址可能会有多个别名，因为不同的指针都可能指向同一个地址
  - 编译器需要知道不同变量是否是别名关系，以便决定能否做某些优化

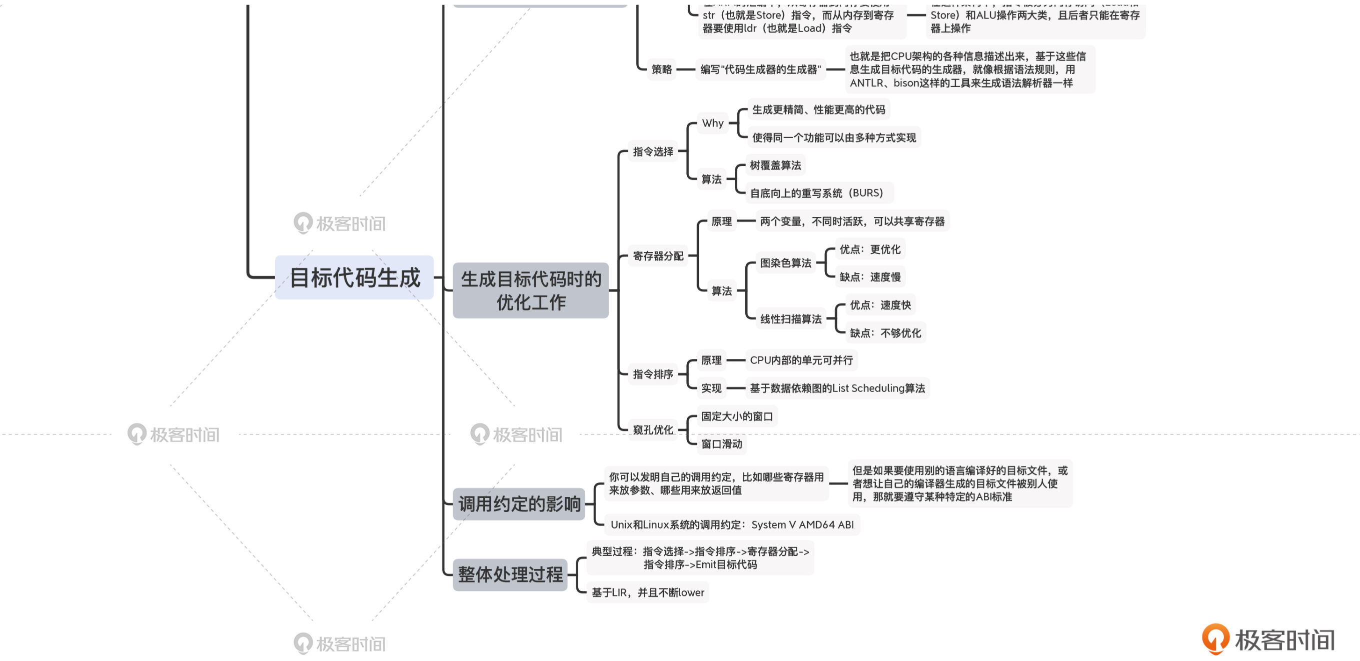
重要性和顺序

- 重要性
  - 对所有语言都重要：循环优化等
  - 面向对象语言：内联、逃逸等
  - 函数式语言：尾递归优化等
- 顺序
  - 要点 — 机器无关-早期，机器相关-后期
  - 注意
    - 一个优化会导致另一个优化
    - 同一个优化会多遍运行

生成针对不同CPU的目标代码

- x86
  - CISC
  - Register-Memory 架构
  - 在指令里可以混合使用内存地址和寄存器
- ARM
  - RISC
  - Load-Store 架构
  - 在ARM的汇编中，从寄存器到内存要使用 … — 在这种架构下，指令被分为内存访问（Load和 …

极客时间



© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 08 | 代码生成：如何实现机器相关的优化？

下一篇 不定期加餐1 | 远程办公，需要你我具备什么样的素质？

写留言

精选留言 (2)

落曦

2020-06-19

太硬核了！

展开

作者回复: 给学习委员点赞！

1

李圣悦

2020-06-19

图片已保存，还要多次学习

展开

作者回复: 赞！